Spring Web MVC

Spring framework makes the development of web applications very easy by providing the Spring MVC module. Spring MVC module is based on two most popular design patterns, they are

- Front controller Design pattern
- MVC Design pattern

Before going into details of Spring MVC architecture, let us first have glance at the two popular design patterns used for web development.

Front Controller design pattern

When the user access the view directly without going thought any centralized mechanism each view is required to provide its; own system services, often resulting in duplication of code and view navigations is left to the views, this may result in commingled view content and view navigation.

A centralized point of contact for handling a request may be useful, for example, to control and log a user's progress through the site. .

Introducing a controller as the initial point of contact for handling a request. The controller manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing, managing the choice of an appropriate view, handling errors, and managing the selection of content creation strategies.

The controller provides a centralized entry point that controls and manages Web request handling. By centralizing decision points and controls, the controller also helps reduce the amount of Java code, called scriptlets, embedded in the JavaServer Pages (JSP) page.

Centralizing control in the controller and reducing business logic in the view promotes code reuse across requests. It is a preferable approach to the alternative-embedding code in multiple views-because that approach may lead to a more error-prone, reuse-by-copy- and-paste environment.

Typically, a controller coordinates with a dispatcher component. Dispatchers are responsible for view management and navigation. Thus, a dispatcher chooses the next view for the user and vectors control to the resource. Dispatchers may be encapsulated within the controller directly or can be extracted into a separate component.

The responsibilities of the components participating in this patterns are:

Controller: The controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.

Dispatcher: A dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource. A dispatcher can be encapsulated within a controller or can be a separate component working in coordination. The dispatcher provides either a static dispatching to the view or a more sophisticated dynamic dispatching mechanism. The dispatcher uses the RequestDispatcher object (supported in the servlet specification) and encapsulates some additional processing.

Helper: A helper is responsible for helping a view or controller complete its processing. Thus, helpers have numerous responsibilities, including gathering data required by the view and storing this intermediate model, in which case the helper is sometimes referred to as a value bean. Additionally, helpers may adapt this data model for use by the view. Helpers can service requests for data from the view by simply providing access to the raw data or by formatting the data as Web content. A view may work with any number of helpers, which are typically implemented as JavaBeans components (JSP 1.0+) and custom tags (JSP 1.1+). Additionally, a helper may represent a Command object, a delegate, or an XSL Transformer, which is used in combination with a stylesheet to adapt and convert the model into the appropriate form.

View: A view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.

Front Controller centralizes control. A controller provides a central place to handle system services and business logic across multiple requests. A controller manages business logic processing and request handling. Centralized access to

an application means that requests are easily tracked and logged. Keep in mind, though, that as control centralizes, it is possible to introduce a single point of failure. In practice, this rarely is a problem, though, since multiple controllers typically exist, either within a single server or in a cluster.

Front Controller improves manageability of security. A controller centralizes control, providing a choke point for illicit access attempts into the Web application. In addition, auditing a single entrance into the application requires fewer resources than distributing security checks across all pages.

Front Controller improves reusability. A controller promotes cleaner application partitioning and encourages reuse, as code that is common among components moves into a controller or is managed by a controller.

Benefits

The following lists the benefits of using the Front Controller pattern:

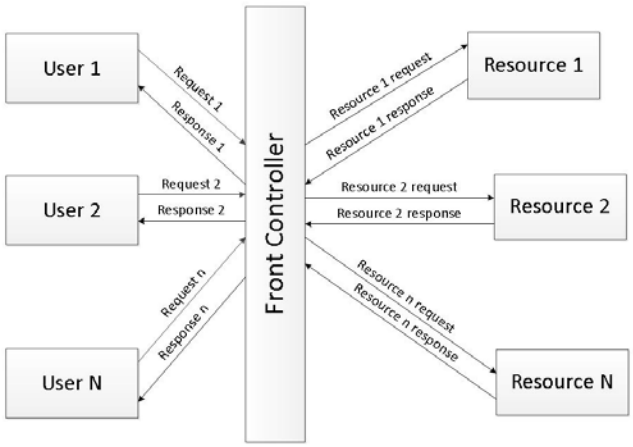
- ✓ Centralizes control logic
- ✓ Improves reusability
- ✓ Improves separation of concerns

When to Use

- ✓ You should use the Front Controller pattern to:
- ✓ Apply common logic to multiple requests
- ✓ Separate processing logic from view

This design pattern enforces a single point of entry for all the incoming requests. All the requests are handled by a single piece of code which can then further delegate the responsibility of processing the request to further application objects.

Front Controller Design pattern



MVC design pattern

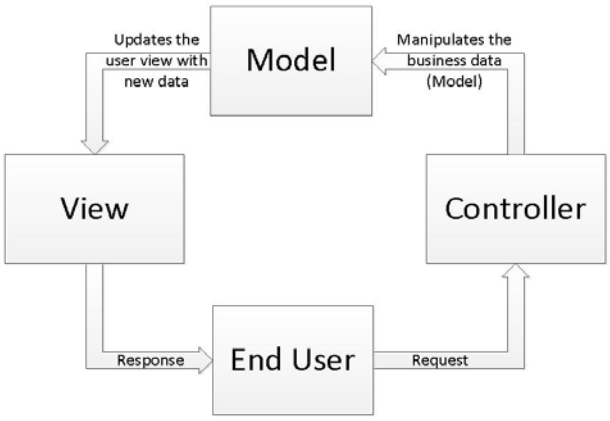
This design pattern helps us develop loosely coupled application by segregating various concerns into different layers. MVC design pattern enforces the application to be divided into three layers, Model, View and Controller.

Model: This represents the application data.

View: This represents the application's user interface. View takes model as the input and renders it appropriately to the end user.

Controller: The controller is responsible for handling the request and generating the model and selecting the appropriate view for the request.

MVC Design Pattern



13.1. Introduction to the spring web MVC framework

Spring's web MVC framework is designed around a **DispatcherServlet** that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest(request,response)method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of, for example AbstractController, AbstractCommandController and SimpleFormController. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: If you don't have a form, you don't need a FormController. This is a major difference to Struts.

You can use any object as a command or form object - there's no need to implement an interface or derive from a base class. Spring's data binding is highly flexible, for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. So you don't need to duplicate your business objects' properties as Strings in your form objects, just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes like Action and ActionForm - for every type of action.

Compared to WebWork, Spring has more differentiated object roles. It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data. Instead, a WebWork Action combines all those roles into one single object. WebWork does allow you to use existing business objects as part of your form, but only by making them bean properties of the respective Action class. Finally, the same Action instance that handles the request is used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the Action too. These are arguably too many roles for one object.

Spring's view resolution is extremely flexible. A Controller implementation can even write a view directly to the response, returning null as ModelAndView. In the normal case, a ModelAndView instance consists of a view name and a model Map, containing bean names and corresponding objects (like a command or form, containing reference data). View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The abstract model Map allows for complete abstraction of the view technology, without any hassle. Any renderer can be integrated directly, whether JSP, Velocity, or any other rendering technology. The model Map is simply transformed into an appropriate format, such as JSP request attributes or a Velocity template model.

13.1.1. Pluggability of other MVC implementations

There are several reasons why some projects will prefer to use other MVC implementations. Many teams expect to leverage their existing investment in skills and tools. In addition, there is a large body of knowledge and experience avalailable for the Struts framework. Thus, if you can live with Struts' architectural flaws, it can still be a viable choice for the web layer. The same applies to WebWork and other web MVC frameworks.
If you don't want to use Spring's web MVC, but intend to leverage other solutions that Spring offers, you can integrate the web MVC framework of your choice with Spring easily. Simply start up a Spring root application context via its ContextLoaderListener, and access it via its ServletContext attribute (or Spring's respective helper method) from within

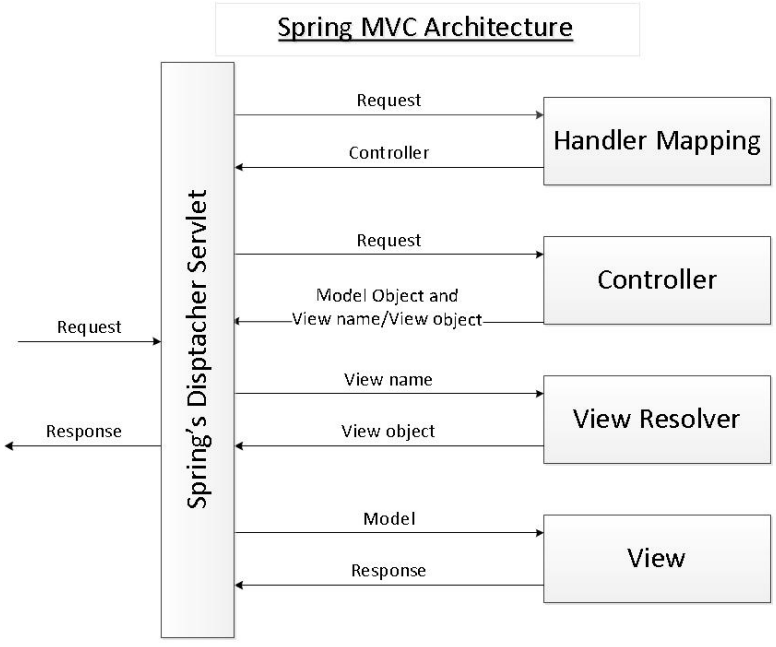
a Struts or WebWork action. Note that there aren't any "plugins" involved, so no dedicated integration is necessary. From the web layer's point of view, you'll simply use Spring as a library, with the root application context instance as the entry point.
All your registered beans and all of Spring's services can be at your fingertips even without Spring's web MVC. Spring doesn't compete with Struts or WebWork in this scenario, it just addresses the many areas that the pure web MVC frameworks don't, from bean configuration to data access and transaction handling. So you are able to enrich your application with a Spring middle tier and/or data access tier, even if you just want to use, for example, the transaction abstraction with JDBC or Hibernate.

13.1.2. Features of Spring MVC

Spring's web module provides a wealth of unique web support features, including:

- **Clear separation of roles -** controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy referencing across contexts, such as from web controllers to business objects and validators.
- **Adaptability, non-intrusiveness -** Use whatever controller subclass you need (plain, command, form, wizard, multi-action, or a custom one) for a given scenario instead of deriving from a single controller for everything.
- **Reusable business code -** no need for duplication. You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.
- **Customizable binding and validation -** type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.
- **Customizable handler mapping and view resolution -** handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. This is more flexible than some web MVC frameworks which mandate a particular technique.
- **Flexible model transfer -** model transfer via a name/value Map supports easy integration with any view technology.
- **Customizable locale and theme resolution**, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.
- **A simple but powerful tag library** - That avoids HTML generation at any cost, allowing for maximum flexibility in terms of markup code.

Spring MVC Architecture



13.2. The DispatcherServlet

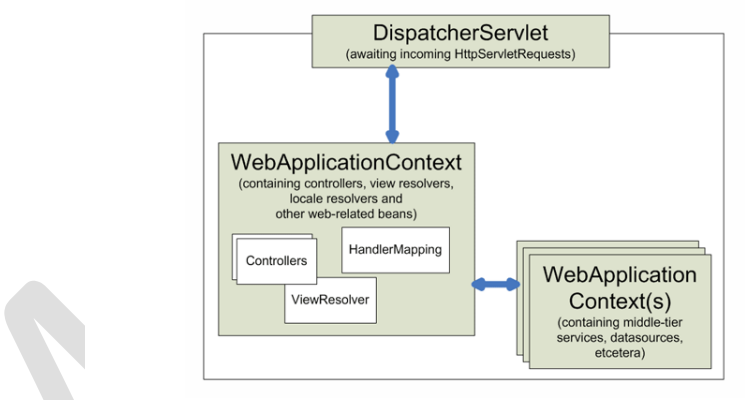
Spring's web MVC framework is, like many other web MVC frameworks, a request-driven web MVC framework, designed around a servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications. Spring's DispatcherServlet however, does more than just that. It is completely integrated with the Spring ApplicationContext and allows you to use every other feature Spring has.

Like ordinary servlets, the DispatcherServlet is declared in the web.xml of your web application. Requests that you want the DispatcherServlet to handle, will have to be mapped, using a URL mapping in the same web.xml file.

```
<web-app>
  ...
  <servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>example</servlet-name>
    <url-pattern>*.form</url-pattern>
  </servlet-mapping>
</web-app>
```

In the example above, all requests ending with .form will be handled by the DispatcherServlet. The DispatcherServlet now needs to be configured.

In the web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in in the Root WebApplicationContext. These inherited beans defined can be overridden in the servlet-specific scope, and new scope-specific beans can be defined local to a given servlet instance.



The framework will, on initialization of a DispatcherServlet, *look for a file named [servlet-name]-servlet.xml* in the WEB-INF directory of your web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

The configLocation used by the DispatcherServlet can be modified through a servlet initialization parameter (see below for details).

The WebApplicationContext is just an ordinary ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that is knows which servlet it is associated with (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and using RequestContextUtils you can always lookup the WebApplicationContext in case you need it.

The Spring DispatcherServlet has a couple of special beans it uses, in order to be able to process requests and render the appropriate views. These beans are included in the Spring framework and can be configured in the WebApplicationContext, just as any other bean would be configured. Each of those beans, is described in more detail below. Right now, we'll just mention them, just to let you know they exist and to enable us to go on talking about the DispatcherServlet. For most of the beans, defaults are provided so you don't have to worry about configuring them.

Table 13.1. Special beans in the WebApplicationContext

Expression	Explanation
handler mapping(s)	A list of pre- and postprocessors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller)
controller(s)	The beans providing the actual functionality (or at least, access to the functionality) as part of the MVC triad
view resolver	Capable of resolving view names to views, used by the DispatcherServlet
locale resolver	Capable of resolving the locale a client is using, in order to be able to offer internationalized views
theme resolver	Capable of resolving themes your web application can use, for example, to offer personalized layouts
multipart resolver	Offers functionality to process file uploads from HTML forms
handlerexception resolver	Offers functionality to map exceptions to views or implement other more complex exception handling code

When a DispatcherServlet is setup for use and a request comes in for that specific DispatcherServlet it starts processing it. The list below describes the complete process a request goes through if handled by a DispatcherServlet:

1. The WebApplicationContext is searched for and bound in the request as an attribute in order for the controller and other elements in the process to use. It is bound by default under the keyDispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE.
2. The locale resolver is bound to the request to let elements in the process resolve the locale to use when processing the request (rendering the view, preparing data, etc.) If you don't use the resolver, it won't affect anything, so if you don't need locale resolving, you don't have to use it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. The theme resolver does not affect anything if you don't use it, so if you don't need themes you can just ignore it.
4. If a multipart resolver is specified, the request is inspected for multiparts and if they are found, it is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process.
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, controllers) will be executed in order to prepare a model.
6. If a model is returned, the view is rendered, using the view resolver that has been configured with the WebApplicationContext. If no model is returned (which could be due to a pre- or postprocessor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

Exceptions that might be thrown during processing of the request get picked up by any of the handlerexception resolvers that are declared in the WebApplicationContext. Using these exception resolvers you can define custom behavior in case such exceptions get thrown.

The Spring DispatcherServlet also has support for returning the *last-modification-date*, as specified by the Servlet API. The process of determining the last modification date for a specific request, is simple. The DispatcherServlet will first lookup an appropriate handler mapping and test if the handler that is found *implements the interface LastModified*and if so, the value of long getLastModified(request) is returned to the client.

You can customize Spring's DispatcherServlet by adding context parameters in the web.xml file or servlet init parameters. The possibilities are listed below.

Table 13.2. DispatcherServlet initialization parameters

Parameter	Explanation
contextClass	Class that implements WebApplicationContext, which will be used to instantiate the context used by this servlet. If this parameter isn't specified, the XmlWebApplicationContext will be used.
contextConfigLocation	String which is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The String is potentially split up into multiple strings (using a comma as a delimiter) to support multiple contexts (in case of multiple context locations, of beans that are defined twice, the latest takes precedence).
namespace	The namespace of the WebApplicationContext. Defaults to [server-name]-servlet.

A typical web application contains both web components & business components

- WebComponents will be handled by WebApplicationContext
- Business components will be handled by ApplicationContext

Configuring ApplicationContext

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootApplicationContext.xml</param-value>
</context-param>
 <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
 </listener>
```

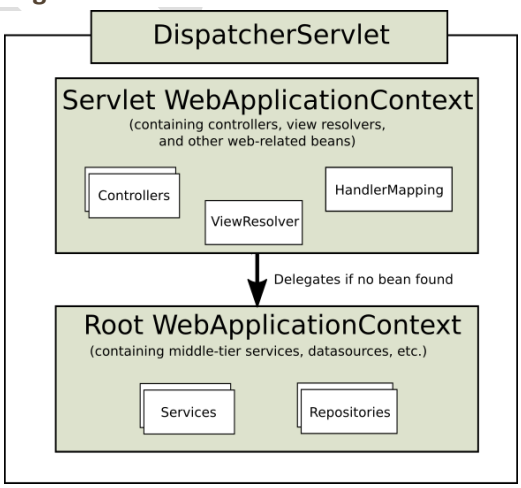
ContextLoaderListener

Performs the actual initialization work for the root application context.

Reads a “contextConfigLocation” context-param and passes its value to the context instance, parsing it into potentially multiple file paths which can be separated by any number of commas and spaces, e.g. “WEB-INF/applicationContext1.xml, WEB-INF/applicationContext2.xml”.

ContextLoaderListener is optional. Just to make a point here: you can boot up a Spring application without ever configuring ContextLoaderListener, just a basic minimum web.xml with DispatcherServlet.

Typical Context hierarchy in Spring Web MVC



13.3. Controllers

The notion of a controller is part of the MVC design pattern. Controllers define application behavior, or at least provide access to the application behavior. Controllers interpret user input and transform the user input into a sensible model which will be represented to the user by the view. Spring has implemented the notion of a controller in a very abstract way enabling a wide variety of different kinds of controllers to be created. Spring contains formcontroller, commandcontroller, controllers that execute wizard-style logic, and more.

Spring's basis for the controller architecture is the org.springframework.web.servlet.mvc.Controller interface, which is listed below.

```
public interface Controller {
   /**
    * Process the request and return a ModelAndView object which the DispatcherServlet
    * will render.
    */
   ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)    throws Exception;
}
```

As you can see, the Controller interface requires a single method that should be capable of handling a request and returning an appropriate model and view. These three concepts are the basis for the Spring MVC implementation - *ModelAndView* and *Controller*. While the Controller interface is quite abstract, Spring offers a lot of controllers that already contain a lot of the functionality you might need. The Controller interface just defines the most common functionality required of every controller - handling a request and returning a model and a view.

13.3.1. AbstractController and WebContentGenerator

Of course, just a controller interface isn't enough. To provide a basic infrastructure, all of Spring's Controllers inherit from AbstractController, a class offering caching support and, for example, the setting of the mimetype.

Table 13.3. Features offered by the AbstractController

Feature	Explanation
supportedMethods	indicates what methods this controller should accept. Usually this is set to both GET and POST, but you can modify this to reflect the method you want to support. If a request is received with a method that is not supported by the controller, the client will be informed of this (using aServletException)).
requiresSession	indicates whether or not this controller requires a session to do its work. This feature is offered to all controllers. If a session is not present when such a controller receives a request, the user is informed using a ServletException.
synchronizeSession	use this if you want handling by this controller to be synchronized on the user's session. To be more specific, extending controller will override the handleRequestInternal method, which will be synchronized if you specify this variable.
cacheSeconds	when you want a controller to generate a caching directive in the HTTP response, specify a positive integer here. By default it is set to *-1* so no caching directives will be included.
useExpiresHeader	tweaks your controllers to specify the HTTP 1.0 compatible *"Expires"* header. By default it's set to true, so you won't have to change it.
useCacheHeader	tweaks your controllers to specify the HTTP 1.1 compatible *"Cache-Control"* header. By default this is set to true so you won't have to change it.

The last two properties are actually part of the WebContentGenerator which is the superclass of AbstractControllerbut are included here for completeness.

When using the AbstractController as a baseclass for your controllers (which is *not* recommended since there are a lot of other controllers that might already do the job for you) you only have to override thehandleRequestInternal(HttpServletRequest, HttpServletResponse) method, implement your logic, and return a ModelAndView object. Here is short example consisting of a class and a declaration in the web application context.

```
package samples;
public class SampleController extends AbstractController {

   public ModelAndView handleRequestInternal(HttpServletRequest request,HttpServletResponse response)
   throws Exception {
      ModelAndView mav = new ModelAndView("foo");
```

```
    mav.addObject("message", "Hello World!");
    return mav;
  }
}

<bean id="sampleController" class="samples.SampleController">
  <property name="cacheSeconds" value="120"/>
</bean>
```

The class above and the declaration in the web application context is all you need besides setting up a handler mapping to get this very simple controller working. This controller will generate caching directives telling the client to cache things for 2 minutes before rechecking. This controller returns an hard-coded view (hmm, not so nice), named index .

13.3.2. Other simple controllers

Although you can extend AbstractController, Spring provides a number of concrete implementations which offer functionality that is commonly used in simple MVC applications. The ParameterizableViewController is basically the same as the example above, except for the fact that you can specify the view name that it will return in the web application context (ahhh, no need to hard-code the viewname).

The UrlFilenameViewController inspects the URL and retrieves the filename of the file request (the filename ofhttp://www.springframework.org/index.html is index) and uses that as a viewname. Nothing more to it.

13.3.3. The MultiActionController

In any medium to large size enterprise web application, there are quite a number of web pages. To fulfil the request for those web pages we need to define multiple controllers, one each for a web page. And sometimes the business logic is executed to fulfil those requests is similar. This creates redundancy of business logic in multiple controllers and makes the maintenance difficult.

Spring's MVC module provides a way to deal with this scenario by providing a single controller fulfilling the request for multiple web pages. Such a controller is known as Multi Action Controller. A user defined multi action controller should extend the class org.springframework.web.servlet.mvc.multiaction.MultiActionController. Each method in user defined multi action controller contains the logic to fulfil the request for a particular web page.

By default, the URL of the incoming request (excluding the extension part) will be matched against the name of the method in multi action controller and the matching method will perform the business logic for the incoming request. So for the incoming request with URL /welcome.htm, the method name containing the business logic will be welcome.

Let us assume that the multi action controller in our application is MyMultiActionController which fulfils the request for the three web pages with URL /welcome.htm, /listBooks.htm and /displayBookTOC.htm. Thus the class should extend the MultiActionController and have three methods with name welcome, listBooks and displayBookTOC. The controller will look as below:

```
public class MyMultiActionController extends MultiActionController {
    // This method will server all the request matching URL pattern /welcome.htm
    public ModelAndView welcome(HttpServletRequest request,
            HttpServletResponse response) {
        // Business logic goes here
        // Return an object of ModelAndView to DispatcherServlet
        return new ModelAndView("Welcome");
    }
    // This method will server all the request matching URL pattern
    // /listBooks.htm
    public ModelAndView listBooks(HttpServletRequest request,
             HttpServletResponse response) {
        // Business logic goes here
        // Return an object of ModelAndView to DispatcherServlet
        return new ModelAndView("listBooks");
    }
    // This method will server all the request matching URL pattern
    // /displayBookTOC.htm
    public ModelAndView displayBookTOC(HttpServletRequest request,
```

```
            HttpServletResponse response) {
        // Business logic goes here
        // Return an object of ModelAndView to DispatcherServlet
        return new ModelAndView("displayBookTOC");
    }
}
```

Table 13.4. Features offered by the MultiActionController

Feature	Explanation
delegate	there are two usage-scenarios for the MultiActionController. Either you subclass the MultiActionController and specify the methods that will be resolved by the MethodNameResolver on the subclass (in which case you don't need to set the delegate), or you define a delegate object, on which methods resolved by the Resolver will be invoked. If you choose this scenario, you will have to define the delegate using this configuration parameter as a collaborator.
methodNameResolver	somehow the MultiActionController will need to resolve the method it has to invoke, based on the request that came in. You can define a resolver that is capable of doing that using this configuration parameter.

Methods defined for a multi-action controller need to conform to the following signature:

```
// actionName can be replaced by any methodname
ModelAndView actionName(HttpServletRequest, HttpServletResponse);
```

Method overloading is not allowed since it would confuse the MultiActionController. Furthermore, you can define *exception handlers* capable of handling exceptions that are thrown by the methods you specify. Exception handler methods need to return a ModelAndView object, just as any other action method and need to conform to the following signature:

```
// anyMeaningfulName can be replaced by any methodname
ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse, ExceptionClass);
```

The ExceptionClass can be *any* exception, as long as it's a subclass of java.lang.Exception orjava.lang.RuntimeException.

The MethodNameResolver is supposed to resolve method names based on the request coming in. There are three resolvers at your disposal, but of course you can implement more of them yourself if you want to.

- ParameterMethodNameResolver - capable of resolving a request parameter and using that as the method name (http://www.sf.net/index.view?testParam=testIt will result in a method testIt(HttpServletRequest, HttpServletResponse) being called). The paramName configuration parameter specifies the parameter that is inspected).
- InternalPathMethodNameResolver - retrieves the filename from the path and uses that as the method name (http://www.sf.net/testing.view will result in a method testing(HttpServletRequest, HttpServletResponse)being called).
- PropertiesMethodNameResolver - uses a user-defined properties object with request URLs mapped to methodnames. When the properties contain /index/welcome.html=doIt and a request to /index/welcome.html comes in, thedoIt(HttpServletRequest, HttpServletResponse) method is called. This method name resolver works with thePathMatcher, so if the properties contained /**/welcom?.html, it would also have worked!

Here are a couple of examples. First, an example showing the ParameterMethodNameResolver and the delegate property, which will accept requests to urls with the parameter method included and set to retrieveIndex:

```
<bean id="paramResolver" class="org....mvc.multiaction.ParameterMethodNameResolver">
  <property name="paramName"><value>method</value></property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
  <property name="methodNameResolver"><ref bean="paramResolver"/></property>
```

```
 <property name="delegate"><ref bean="sampleDelegate"/></property>
</bean>
<bean id="sampleDelegate" class="samples.SampleDelegate"/>

## together with
public class SampleDelegate {
   public ModelAndView retrieveIndex(HttpServletRequest req,   HttpServletResponse resp) {
      return new ModelAndView("index", "date", new Long(System.currentTimeMillis()));
   }
}
```

When using the delegates shown above, we could also use the PropertiesMethodNameResolver to match a couple of URLs to the method we defined:

```
<bean id="propsResolver" class="org....mvc.multiaction.PropertiesMethodNameResolver">
 <property name="mappings">
  <props>
   <prop key="/index/welcome.html">retrieveIndex</prop>
   <prop key="/**/notwelcome.html">retrieveIndex</prop>
   <prop key="/*/user?.html">retrieveIndex</prop>
  </props>
 </property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
  <property name="methodNameResolver"><ref bean="propsResolver"/></property>
  <property name="delegate"><ref bean="sampleDelegate"/></property>
</bean>
```

13.3.4. CommandControllers

Spring's *CommandControllers* are a fundamental part of the Spring MVC package. Command controllers provide a way to interact with data objects and dynamically bind parameters from the HttpServletRequest to the data object specified. They perform a similar role to Struts' ActionForm, but in Spring, your data objects don't have to implement a framework-specific interface. First, let's examine what command controllers available, to get overview of what you can do with them:

- AbstractCommandController - a command controller you can use to create your own command controller, capable of binding request parameters to a data object you specify. This class does not offer form functionality, it does however, offer validation features and lets you specify in the controller itself what to do with the command object that has been filled with the parameters from the request.
- AbstractFormController - an abstract controller offering form submission support. Using this controller you can model forms and populate them using a command object you retrieve in the controller. After a user has filled the form, the AbstractFormController binds the fields, validates, and hands the object back to the controller to take appropriate action. Supported features are: invalid form submission (resubmission), validation, and normal form workflow. You implement methods to determine which views are used for form presentation and success. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the application context.
- SimpleFormController - a concrete FormController that provides even more support when creating a form with a corresponding command object. The SimpleFormController let's you specify a command object, a viewname for the form, a viewname for page you want to show the user when form submission has succeeded, and more.
- AbstractWizardFormController - as the class name suggests, this is an abstract class--your WizardController should extend it. This means you have to implement the validatePage(), processFinish and processCancelmethods.
  You probably also want to write a contractor, which should at the very least call setPages() andsetCommandName(). The former takes as its argument an array of type String. This array is the list of views which comprise your wizard. The latter takes as its argument a String, which will be used to refer to your command object from within your views.

As with any instance of AbstractFormController, you are required to use a command object - a JavaBean which will be populated with the data from your forms. You can do this in one of two ways: either call setCommandClass()from the constructor with the class of your command object, or implement the formBackingObject() method.

AbstractWizardFormController has a number of concrete methods that you may wish to override. Of these, the ones you are likely to find most useful are: referenceData which you can use to pass model data to your view in the form of a Map; getTargetPage if your wizard needs to change page order or omit pages dynamically; andonBindAndValidate if you want to override the built-in binding and validation workflow.

Finally, it is worth pointing out the setAllowDirtyBack and setAllowDirtyForward, which you can call fromgetTargetPage to allow users to move backwards and forwards in the wizard even if validation fails for the current page.

For a full list of methods, see the JavaDoc for AbstractWizardFormController. There is an implemented example of this wizard in the jPetStore included in the Spring distribution: org.springframework.samples.jpetstore.web.spring.OrderFormController.

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @RequestMapping, @RequestParam, @ModelAttribute, and so on. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet APIs, although you can easily configure access to Servlet or Portlet facilities.

```
@Controller
public class HelloWorldController {

  @RequestMapping("/helloWorld")
  public String helloWorld(Model model) {
    model.addAttribute("message", "Hello World!");
    return "helloWorld";
  }
}
```

As you can see, the @Controller and @RequestMapping annotations allow flexible method names and signatures. In this particular example the method accepts a Model and returns a view name as a String, but various other method parameters and return values can be used as explained later in this section. @Controller and @RequestMapping and a number of other annotations form the basis for the Spring MVC implementation. This section documents these annotations and how they are most commonly used in a Servlet environment.

The @Controller annotation indicates that a particular class serves the role of a *controller*. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations (see the next section).

You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the *spring-context* schema as shown in the following XML snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
```

```
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <context:component-scan base-package="org.springframework.samples.petclinic.web"/>

   <!-- ... -->
</beans>
```

Mapping Requests With @RequestMapping

You use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

The following example from the Petcare sample shows a controller in a Spring MVC application that uses this annotation:

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

   private final AppointmentBook appointmentBook;

   @Autowired
   public AppointmentsController(AppointmentBook appointmentBook) {
      this.appointmentBook = appointmentBook;
   }

   @RequestMapping(method = RequestMethod.GET)
   public Map<String, Appointment> get() {
      return appointmentBook.getAppointmentsForToday();
   }
   @RequestMapping(path = "/{day}", method = RequestMethod.GET)
   public Map<String, Appointment> getForDay(@PathVariable   @DateTimeFormat(iso=ISO.DATE) Date day, Model
model) {
      return appointmentBook.getAppointmentsForDay(day);
   }

   @RequestMapping(path = "/new", method = RequestMethod.GET)
   public AppointmentForm getNewForm() {
      return new AppointmentForm();
   }

   @RequestMapping(method = RequestMethod.POST)
   public String add(@Valid AppointmentForm appointment, BindingResult result) {
      if (result.hasErrors()) {
         return "appointments/new";
      }
      appointmentBook.addAppointment(appointment);
      return "redirect:/appointments";
   }
}
```

13.4. Handler mappings

Using a handler mapping you can map incoming web requests to appropriate handlers. There are some handler mappings you can use out of the box, for example, the SimpleUrlHandlerMapping or the BeanNameUrlHandlerMapping, but let's first examine the general concept of a HandlerMapping.

The functionality a basic HandlerMapping provides is the delivering of a HandlerExecutionChain, which must contain the handler that matches the incoming request, and may also contain a list of handler interceptors that are applied to the request. When a request comes in, the DispatcherServlet will hand it over to the handler mapping to let it inspect the request and come up with an appropriate HandlerExecutionChain. Then the DispatcherServlet will execute the handler and interceptors in the chain (if any).

The concept of configurable handler mappings that can optionally contain interceptors (executed before or after the actual handler was executed, or both) is extremely powerful. A lot of supporting functionality can be built into custom HandlerMappings. Think of a custom handler mapping that chooses a handler not only based on the URL of the request coming in, but also on a specific state of the session associated with the request.
This section describes two of Spring's most commonly used handler mappings. They both extend the AbstractHandlerMapping and share the following properties:

- interceptors: the list of interceptors to use. HandlerInterceptors are discussed in Section 13.4.3, "AddingHandlerInterceptors".
- defaultHandler: the default handler to use, when this handler mapping does not result in a matching handler.
- order: based on the value of the order property (see the org.springframework.core.Ordered interface), Spring will sort all handler mappings available in the context and apply the first matching handler.
- alwaysUseFullPath: if this property is set to true, Spring will use the full path within the current servlet context to find an appropriate handler. If this property is set to false (the default), the path within the current servlet mapping will be used. For example, if a servlet is mapped using /testing/* and thealwaysUseFullPath property is set to true, /testing/viewPage.html would be used, whereas if the property is set to false, /viewPage.html would be used.
- urlPathHelper: using this property, you can tweak the UrlPathHelper used when inspecting URLs. Normally, you shouldn't have to change the default value.
- urlDecode: the default value for this property is false. The HttpServletRequest returns request URLs and URIs that are *not* decoded. If you do want them to be decoded before a HandlerMapping uses them to find an appropriate handler, you have to set this to true (note that this requires JDK 1.4). The decoding method uses either the encoding specified by the request or the default ISO-8859-1 encoding scheme.
- lazyInitHandlers: allows for lazy initialization of *singleton* handlers (prototype handlers are always lazily initialized). Default value is false.

(*Note: the last four properties are only available to subclasses of org.springframework.web.servlet.handler.AbstractUrlHandlerMapping*).

13.4.1. BeanNameUrlHandlerMapping

A very simple, but very powerful handler mapping is the BeanNameUrlHandlerMapping, which maps incoming HTTP requests to names of beans, defined in the web application context. Let's say we want to enable a user to insert an account and we've already provided an appropriate FormController (see Section 13.3.4, "CommandControllers" for more information on Command- and FormControllers) and a JSP view (or Velocity template) that renders the form.
When using the BeanNameUrlHandlerMapping, we could map the HTTP request with
URL http://samples.com/editaccount.form to the appropriate FormController as follows:

```
<beans>
 <bean id="handlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

 <bean name="/editaccount.form" class="org.springframework.web.servlet.mvc.SimpleFormController">
  <property name="formView"><value>account</value></property>
  <property name="successView"><value>account-created</value></property>
  <property name="commandName"><value>Account</value></property>
  <property name="commandClass"><value>samples.Account</value></property>
 </bean>
<beans>
```

All incoming requests for the URL /editaccount.form will now be handled by the FormController in the source listing above. Of course we have to define a servlet-mapping in web.xml as well, to let through all the requests ending with .form.

```
<web-app>
   ...
   <servlet>
      <servlet-name>sample</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

  <!-- Maps the sample dispatcher to /*.form -->
   <servlet-mapping>
      <servlet-name>sample</servlet-name>
      <url-pattern>*.form</url-pattern>
   </servlet-mapping>
   ...
</web-app>
```

NOTE: if you want to use the BeanNameUrlHandlerMapping, you don't necessarily have to define it in the web application context (as indicated above). By default, if no handler mapping can be found in the context, the DispatcherServlet creates a BeanNameUrlHandlerMapping for you!

13.4.2. SimpleUrlHandlerMapping

A further - and much more powerful handler mapping - is the SimpleUrlHandlerMapping. This mapping is configurable in the application context and has Ant-style path matching capabilities (see the JavaDoc fororg.springframework.util.PathMatcher). Here is an example:

```
<web-app>
   ...
   <servlet>
      <servlet-name>sample</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <!-- Maps the sample dispatcher to /*.form -->
   <servlet-mapping>
      <servlet-name>sample</servlet-name>
      <url-pattern>*.form</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>sample</servlet-name>
      <url-pattern>*.html</url-pattern>
   </servlet-mapping>
   ...
</web-app>
```

Allows all requests ending with .html and .form to be handled by the sample dispatcher servlet.

```
<beans>
   <bean id="handlerMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
      <property name="mappings">
         <props>
            <prop key="/*/account.form">editAccountFormController</prop>
            <prop key="/*/editaccount.form">editAccountFormController</prop>
            <prop key="/ex/view*.html">someViewController</prop>
            <prop key="/**/help.html">helpController</prop>
```

```
        </props>
      </property>
  </bean>

  <bean id="someViewController"
      class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

  <bean id="editAccountFormController"
      class="org.springframework.web.servlet.mvc.SimpleFormController">
    <property name="formView"><value>account</value></property>
    <property name="successView"><value>account-created</value></property>
    <property name="commandName"><value>Account</value></property>
    <property name="commandClass"><value>samples.Account</value></property>
  </bean>
<beans>
```

This handler mapping routes requests for help.html in any directory to the helpController, which is a UrlFilenameViewController (more about controllers can be found in Section 13.3, "Controllers"). Requests for a resource beginning with view, and ending with .html in the directory ex, will be routed to the someViewController. Two further mappings are defined for editAccountFormController.

13.4.3. Adding HandlerInterceptors

Spring's handler mapping mechanism has a notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal.

Interceptors located in the handler mapping must implement HandlerInterceptor from theorg.springframework.web.servlet package. This interface defines three methods, one that will be called *before* the actual handler will be executed, one that will be called *after* the handler is executed, and one that is called *after the complete request has finished*. These three methods should provide enough flexibility to do all kinds of pre- and post-processing.

The preHandle method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue, when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

The following example provides an interceptor that intercepts all requests and reroutes the user to a specific page if the time is not between 9 a.m. and 6 p.m.

```
<beans>
  <bean id="handlerMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
      <list>
        <ref bean="officeHoursInterceptor"/>
      </list>
    </property>
    <property name="mappings">
      <props>
        <prop key="/*.form">editAccountFormController</prop>
        <prop key="/*.view">editAccountFormController</prop>
      </props>
    </property>
  </bean>

  <bean id="officeHoursInterceptor"
      class="samples.TimeBasedAccessInterceptor">
    <property name="openingTime"><value>9</value></property>
    <property name="closingTime"><value>18</value></property>
  </bean>
```

```
<beans>

package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

   private int openingTime;
   private int closingTime;
   public void setOpeningTime(int openingTime) {
      this.openingTime = openingTime;
   }
   public void setClosingTime(int closingTime) {
      this.closingTime = closingTime;
   }
   public boolean preHandle(HttpServletRequest request,HttpServletResponse response, Object handler)
   throws Exception {
      Calendar cal = Calendar.getInstance();
      int hour = cal.get(HOUR_OF_DAY);
      if (openingTime <= hour < closingTime) {
         return true;
      } else {
         response.sendRedirect("http://host.com/outsideOfficeHours.html");
         return false;
      }
   }
}
```

Any request coming in, will be intercepted by the TimeBasedAccessInterceptor, and if the current time is outside office hours, the user will be redirected to a static html file, saying, for example, he can only access the website during office hours.
As you can see, spring has an adapter to make it easy for you to extend the HandlerInterceptor.

13.4.4 RequestMappingHandlerMapping

This was introduced in spring 3.1. Earlier the DefaultAnnotationHandlerMapping decided which controller to use and the AnnotationMethodHandlerAdapter selected the actual method that handled the request. RequestMappingHandlerMapping does both the tasks. Therefore the request is directly mapped right to the method. The following types can be passed to method handlers.

```
@Controller
public class WelcomeController {
 @RequestMapping("/HelloWorld")
 public ModelAndView Welcome()
        String message = "Welcome to Spring  MVC Tutorial";
        //returing viewName,messageKey,messageVariable
        return new ModelAndView("Welcome","message",message);
 }
}
```

13.5. Views and resolving them

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use Java Server Pages, Velocity templates and XSLT views, for example. Chapter 14, *Integrating view technologies* has details of integrating various view technologies.
The two interfaces which are important to the way Spring handles views are ViewResolver and View.
The ViewResolverprovides a mapping between view names and actual views. The View interface addresses the preparation of the request and hands the request over to one of the view technologies.

13.5.1. ViewResolvers

As discussed
all controllers in the Spring web MVC framework, return a ModelAndViewinstance. Views in Spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. We'll list most of them and then provide a couple of examples.

Table 13.5. View resolvers

ViewResolver	Description
AbstractCachingViewResolver	An abstract view resolver which takes care of caching views. Often views need preparation before they can be used, extending this view resolver provides caching of views.
XmlViewResolver	An implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's bean factories. The default configuration file is /WEB-INF/views.xml.
ResourceBundleViewResolver	An implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. The default file name is views.properties.
UrlBasedViewResolver	A simple implementation of ViewResolver that allows for direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
InternalResourceViewResolver	A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses like JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass. See UrlBasedViewResolver's javadocs for details.
VelocityViewResolver / FreeMarkerViewResolver	A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them.

As an example, when using JSP for a view technology you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over the RequestDispatcher to render the view.

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="prefix"><value>/WEB-INF/jsp/</value></property>
  <property name="suffix"><value>.jsp</value></property>
</bean>
```

When returning test as a viewname, this view resolver will hand the request over to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp.
When mixing different view technologies in a web application, you can use the ResourceBundleViewResolver:

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename"><value>views</value></property>
  <property name="defaultParentView"><value>parentView</value></property>
</bean>
```

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].class as the view class and the value of the property[viewname].url as the view url. As you can see, you can identify a parent view, from which all views in the properties file sort of extend. This way you can specify a default view class, for example.

A note on caching - subclasses of AbstractCachingViewResolver cache view instances they have resolved. This greatly improves performance when using certain view technology. It's possible to turn off the cache, by setting the cacheproperty to false. Furthermore, if you have the requirement to be able to refresh a certain view at runtime (for example when a Velocity template has been modified), you can use the removeFromCache(String viewName, Locale loc) method.

13.5.2. Chaining ViewResolvers

Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances. Chaining view resolvers is pretty straightforward - just add more than one resolver to your application context and, if necessary, set the order property to specify an order. Remember, the higher the order property, the later the view resolver will be positioned in the chain.

In the following example, the chain of view resolvers consists of two resolvers, a InternalResourceViewResolver (which is always automatically positioned as the last resolver in the chain) and an XmlViewResolver for specifying Excel views (which are not supported by the InternalResourceViewResolver):

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

### views.xml
<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

If a specific view resolver does not result in a view, Spring will inspect the context to see if other view resolvers are configured. If there are additional view resolvers, it will continue to inspect them. If not, it will throw an Exception.

You have to keep something else in mind - the contract of a view resolver mentions that a view resolver *can* return null to indicate the view could not be found. Not all view resolvers do this however! This is because in some cases, the resolver simply cannot detect whether or not the view exists. For example, the InternalResourceViewResolver uses the RequestDispatcher internally, and dispatching is the only way to figure out if a JSP exists -this can only be done once. The same holds for the VelocityViewResolver and some others. Check the JavaDoc for the view resolver to see if you're dealing with a view resolver that does not report non-existing views. As a result of this, putting anInternalResourceViewResolver in the chain in a place other than the last, will result in the chain not being fully inspected, since the InternalResourceViewResolver will *always* return a view!

13.5.3. Redirecting to views

As has been mentioned, a controller normally returns a logical view name, which a view resolver resolves to a particular view technology. For view tehcnologies such as JSPs that are actually processed via the Servlet/JSP engine, this is normally handled via InternalResourceViewResolver/InternalResourceView which will ultimately end up issuing an internal forward or include, via the Servlet API's RequestDispatcher.forward() or RequestDispatcher.include(). For other view technologies, such as Velocity, XSLT, etc., the view itself produces the content on the response stream.

It is sometimes desireable to issue an HTTP redirect back to the client, before the view is rendered. This is desireable for example when one controller has been called with POSTed data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean the other controller will also see the same POST data, whih is potentially problematic if it can confuse it with other expected data Another reason to do a redirect before displaying the result is that this will eliminate the possiblity of the user doing a double submission of form data. The browser will have sent the initial POST, will have seen a redirect back and done a subsequent GET because of that, and thus as far as it is concerned, the current page does not reflect the result of a POST, but rather of a GET, so there is no way the user can accidentally re-POST the same data by doing a refresh. The refresh would just force a GET of the result page, not a resend of the initial POST data.

13.5.3.1. RedirectView

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring's RedirectView. In this case, DispatcherServlet will not use the normal view resolution mechanism, but rather as it has been given the (redirect) view already, will just ask it to do it's work.
The RedirectView simply ends up issuing an HttpServletResponse.sendRedirect() call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.
If using RedirectView, and the view is created by the Controller itself, it is generally always preferable if the redirect URL at least is injected into the Controller, so that it is not baked into the controller but rather configured in the context along with view names and the like.

13.5.3.2. The redirect: prefix

While the use of RedirectView works fine, if the controller itself is creating the RedirectView, there is no getting around the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled. It should generally think only in terms of view names, that have been injected into it.

The special redirect: prefix allows this to be achived. If a view name is returned which has the prefix redirect:, thenUrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can deal just in terms of logical view names. A logical view name such as redirect:/my/response/controller.html will redirect relative to the current servlet context, while a name such as redirect:http://myhost.com/some/arbitrary/path.htmlwill redirect to an absolute URL. The important thing is that as long is this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

13.5.3.3. The forward: prefix

It is also possible to use a special forward: prefix for view names that will ultimately be resolved byUrlBasedViewResolver and subclasses. All this does is create an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using InternalResourceViewResolver/InternalResourceView anyway (for JSPs for example), but it's of potential use when you are primarilly using another view technology, but want to still be able to in some cases force a forward to happen to a resource to be handled by the Servlet/JSP engine. Note that if you need to do this a lot though, you may also just chain multiple view resolvers. As with the redirect: prefix, if the view name with the prefix is just injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.

13.6. Using locales

Most parts of Spring's architecture support internationalization, just as the Spring web MVC framework does. DispatcherServlet enables you to automatically resolve messages using the client's locale. This is done with LocaleResolver objects.
When a request comes in, the DispatcherServlet looks for a locale resolver and if it finds one it tries to use it to set the locale. Using the RequestContext.getLocale() method, you can always retrieve the locale that was resolved by the locale resolver.
Besides the automatic locale resolution, you can also attach an interceptor to the handler mapping (see Section 13.4.3, “Adding HandlerInterceptors” for more information on handler mapping interceptors), to change the locale under specific circumstances, based on a parameter in the request, for example.
Locale resolvers and interceptors are all defined in the org.springframework.web.servlet.i18n package, and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

13.6.1. AcceptHeaderLocaleResolver

This locale resolver inspects the accept-language header in the request that was sent by the browser of the client. Usually this header field contains the locale of the client's operating system.

13.6.2. CookieLocaleResolver

This locale resolver inspects a Cookie that might exist on the client, to see if a locale is specified. If so, it uses that specific locale. Using the properties of this locale resolver, you can specify the name of the cookie, as well as the maximum age.

```
<bean id="localeResolver">
   <property name="cookieName"><value>clientlanguage</value></property>
   <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
   <property name="cookieMaxAge"><value>100000</value></property>
</bean>
```

This is an example of defining a CookieLocaleResolver.

Table 13.6. Special beans in the WebApplicationContext

Property	Default	Description
cookieName	classname + LOCALE	The name of the cookie
cookieMaxAge	Integer.MAX_INT	The maximum time a cookie will stay persistent on the client. If -1 is specified, the cookie will not be persisted. It will only be available until the client shuts down his or her browser.
cookiePath	/	Using this parameter, you can limit the visibility of the cookie to a certain part of your site. When cookiePath is specified, the cookie will only be visible to that path, and the paths below it.

13.6.3. SessionLocaleResolver

The SessionLocaleResolver allows you to retrieve locales from the session that might be associated with the user's request.

13.6.4. LocaleChangeInterceptor

You can build in changing of locales using the LocaleChangeInterceptor. This interceptor needs to be added to one of the handler mappings (see Section 13.4, “Handler mappings”). It will detect a parameter in the request and change the locale (it calls setLocale() on the LocaleResolver that also exists in the context).

```
<bean id="localeChangeInterceptor"
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
   <property name="paramName"><value>siteLanguage</value></property>
</bean>

<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
   <property name="interceptors">
      <list>
         <ref bean="localeChangeInterceptor"/>
      </list>
   </property>
   <property name="mappings">
      <props>
         <prop key="/**/*.view">someController</prop>
      </props>
   </property>
</bean>
```

All calls to all *.view resources containing a parameter named siteLanguage will now change the locale. So a call tohttp://www.sf.net/home.view?siteLanguage=nl will change the site language to Dutch.

13.7. Using themes

13.7.1. Introduction

The *theme* support provided by the Spring web MVC framework enables you to further enhance the user experience by allowing the look and feel of your application to be *themed*. A theme is basically a collection of static resources affecting the visual style of the application, typically style sheets and images.

13.7.2. Defining themes

When you want to use themes in your web application you'll have to setup a org.springframework.ui.context.ThemeSource. The WebApplicationContext interface extends ThemeSource but delegates its responsabilities to a dedicated implementation. By default the delegate will be aorg.springframework.ui.context.support.ResourceBundleThemeSource that loads properties files from the root of the classpath. If you want to use a custom ThemeSource implementation or if you need to configure the basename prefix of the ResourceBundleThemeSource, you can register a bean in the application context with the reserved name "themeSource". The web application context will automatically detect that bean and start using it.
When using the ResourceBundleThemeSource, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme. Here's an example:

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

The keys of the properties are the names used to refer to the themed elements from view code. For a JSP this would typically be done using the spring:theme custom tag, which is very similar to the spring:message tag. The following JSP fragment uses the theme defined above to customize the look and feel:

```
<taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
  <head>
    <link rel="stylesheet" href="<spring:theme code="styleSheet"/>" type="text/css"/>
  </head>
  <body background="<spring:theme code="background"/>">
    ...
  </body>
</html>
```

By default, the ResourceBundleThemeSource uses an empty basename prefix. As a result the properties files will be loaded from the root of the classpath, so we'll have to put our cool.properties theme definition in a directory at the root of the classpath, e.g. in /WEB-INF/classes. Note that the ResourceBundleThemeSource uses the standard Java resource bundle loading mechanism, allowing for full internationalisation of themes. For instance, we could have a /WEB-INF/classes/cool_nl.properties that references a special background image, e.g. with Dutch text on it.

13.7.3. Theme resolvers

Now that we have our themes defined, the only thing left to do is decide which theme to use.
The DispatcherServlet will look for a bean named "themeResolver" to find out which ThemeResolver implementation to use. A theme resolver works in much the same way as a LocalResolver. It can detect the theme that should be used for a particular request and can also alter the request's theme. The following theme resolvers are provided by Spring:

Table 13.7. ThemeResolver implementations

Class	Description
FixedThemeResolver	Selects a fixed theme, set using the "defaultThemeName" property.
SessionThemeResolver	The theme is maintained in the users HTTP session. It only needs to be set once for each session, but is not persisted between sessions.
CookieThemeResolver	The selected theme is stored in a cookie on the client's machine.

Spring also provides a ThemeChangeInterceptor, which allows changing the theme on every request by including a simple request parameter.

13.8. Spring's multipart (fileupload) support

13.8.1. Introduction

Spring has built-in multipart support to handle fileuploads in web applications. The design for the multipart support is done with pluggable MultipartResolver objects, defined in the org.springframework.web.multipart package. Out of the box, Spring provides MultipartResolvers for use with *Commons FileUpload* (http://jakarta.apache.org/commons/fileupload) and *COS FileUpload* (http://www.servlets.com/cos). How uploading files is supported will be described in the rest of this chapter.

By default, no multipart handling will be done by Spring, as some developers will want to handle multiparts themselves. You will have to enable it yourself by adding a multipart resolver to the web application's context. After you have done that, each request will be inspected to see if it contains a multipart. If no multipart is found, the request will continue as expected. However, if a multipart is found in the request, the MultipartResolver that has been declared in your context will be used. After that, the multipart attribute in your request will be treated like any other attribute.

13.8.2. Using the MultipartResolver

The following example shows how to use the CommonsMultipartResolver:

```
<bean id="multipartResolver"
   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

   <!-- one of the properties available; the maximum file size in bytes -->
   <property name="maxUploadSize">
      <value>100000</value>
   </property>
</bean>
```

This is an example using the CosMultipartResolver:

```
<bean id="multipartResolver"
   class="org.springframework.web.multipart.cos.CosMultipartResolver">

   <!-- one of the properties available; the maximum file size in bytes -->
   <property name="maxUploadSize">
      <value>100000</value>
   </property>
</bean>
```

Of course you need to stick the appropriate jars in your classpath for the multipart resolver to work. In the case of the CommonsMultipartResolver, you need to use commons-fileupload.jar, while in the case of the CosMultipartResolver, use cos.jar.

Now that you have seen how to set Spring up to handle multipart requests, let's talk about how to actually use it. When the Spring DispatcherServlet detects a Multipart request, it activates the resolver that has been declared in your context and hands over the request. What it basically does is wrap the current HttpServletRequest into aMultipartHttpServletRequest that has support for multiparts. Using the MultipartHttpServletRequest you can get information about the multiparts contained by this request and actually get the multiparts themselves in your controllers.

13.8.3. Handling a fileupload in a form

After the MultipartResolver has finished doing its job, the request will be processed like any other. To use it, you create a form with an upload field, then let Spring bind the file on your form. Just as with any other property that's not automagically convertible to a String or primitive type, to be able to put binary data in your beans you have to register a custom editor with the ServletRequestDatabinder. There are a couple of editors available for handling files and setting the results on a bean. There's a StringMultipartEditor capable of converting files to Strings (using a user-defined character set) and there is a ByteArrayMultipartEditor which converts files to byte arrays. They function just as the CustomDateEditor does.

So, to be able to upload files using a form in a website, declare the resolver, a url mapping to a controller that will process the bean, and the controller itself.

```
<beans>
```

```
   ...
   <bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
   <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
         <props>
            <prop key="/upload.form">fileUploadController</prop>
         </props>
      </property>
   </bean>

   <bean id="fileUploadController" class="examples.FileUploadController">
      <property name="commandClass"><value>examples.FileUploadBean</value></property>
      <property name="formView"><value>fileuploadform</value></property>
      <property name="successView"><value>confirmation</value></property>
   </bean>

</beans>
```

After that, create the controller and the actual bean to hold the file property

```
// snippet from FileUploadController
public class FileUploadController extends SimpleFormController {

   protected ModelAndView onSubmit(
      HttpServletRequest request,
      HttpServletResponse response,
      Object command,
      BindException errors)
      throws ServletException, IOException {
      // cast the bean
      FileUploadBean bean = (FileUploadBean)command;
      // let's see if there's content there
      byte[] file = bean.getFile();
      if (file == null) {
         // hmm, that's strange, the user did not upload anything
      }

      // well, let's do nothing with the bean for now and return:
      return super.onSubmit(request, response, command, errors);
   }

   protected void initBinder(  HttpServletRequest request, ServletRequestDataBinder binder)
      throws ServletException {
      // to actually be able to convert Multipart instance to byte[]
      // we have to register a custom editor (in this case the
      // ByteArrayMultipartEditor
      binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
      // now Spring knows how to handle multipart object and convert them
   }

}
// snippet from FileUploadBean
public class FileUploadBean {
   private byte[] file;
   public void setFile(byte[] file) {
      this.file = file;
```

```
    }
    public byte[] getFile() {
        return file;
    }
}
```

As you can see, the FileUploadBean has a property typed byte[] that holds the file. The controller registers a custom editor to let Spring know how to actually convert the multipart objects the resolver has found to properties specified by the bean. In these examples, nothing is done with the byte[] property of the bean itself, but in practice you can do whatever you want (save it in a database, mail it to somebody, etc).

But we're still not finished. To actually let the user upload something, we have to create a form:

```
<html>
    <head>
        <title>Upload a file please</title>
    </head>
    <body>
        <h1>Please upload a file</h1>
        <form method="post" action="upload.form" enctype="multipart/form-data">
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

As you can see, we've created a field named after the property of the bean that holds the byte[]. Furthermore we've added the encoding attribute which is necessary to let the browser know how to encode the multipart fields (do not forget this!). Now everything should work.

13.9. Handling exceptions

Spring provides HandlerExceptionResolvers to ease the pain of unexpected exceptions occurring while your request is being handled by a controller which matched the request. HandlerExceptionResolvers somewhat resemble the exception mappings you can define in the web application descriptor web.xml. However, they provide a more flexible way to handle exceptions. They provide information about what handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exception gives you many more options for how to respond appropriately before the request is forwarded to another URL (the same end result as when using the servlet specific exception mappings).

Besides implementing the HandlerExceptionResolver, which is only a matter of implementing the resolveException(Exception, Handler) method and returning a ModelAndView, you may also use the SimpleMappingExceptionResolver. This resolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it's also possible to implement more fine grained mappings of exceptions from different handlers.

Chapter 14. Integrating view technologies

14.1. Introduction

One of the areas in which Spring excels is in the separation of view technologies from the rest of the MVC framework. For example, deciding to use Velocity or XSLT in place of an existing JSP is primarily a matter of configuration. This chapter covers the major view technologies that work with Spring and touches briefly on how to add new ones. This chapter assumes you are already familiar with Section 13.5, “Views and resolving them” which covers the basics of how views in general are coupled to the MVC framework.

14.2. JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. Using JSP or JSTL is done using a normal viewresolver defined in the WebApplicationContext. Furthermore, of course you need to write some JSPs that will actually render the view. This part describes some of the additional features Spring provides to facilitate JSP development.

14.2.1. View resolvers

Just as with any other view technology you're integrating with Spring, for JSPs you'll need a view resolver that will resolve your views. The most commonly used view resolvers when developing with JSPs are the InternalResourceViewResolver and the ResourceBundleViewResolver. Both are declared in the WebApplicationContext:

```
# The ResourceBundleViewResolver:
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):
welcome.class=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

productList.class=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

As you can see, the ResourceBundleViewResolver needs a properties file defining the view names mapped to 1) a class and 2) a URL. With a ResourceBundleViewResolver you can mix different types of views using only one resolver.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

The InternalResourceBundleViewResolver can be configured for using JSPs as described above. As a best practice, we strongly encourage placing your JSP files in a a directory under the WEB-INF directory, so there can be no direct access by clients.

14.2.2. 'Plain-old' JSPs versus JSTL

When using Java Standard Tag Library you must use a special view class, the JstlView, as JSTL needs some preparation before things such as the i18N features will work.

14.2.3. Additional tags facilitating development

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags have *html escaping* features to enable or disable escaping of characters.
The tag library descriptor (TLD) is included in the spring.jar as well in the distribution itself. More information about the individual tags can be found online: http://www.springframework.org/docs/taglib/index.html.

14.3. Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.

14.3.1. Dependencies

To be able to use Tiles you have to have a couple of additional dependencies included in your project. The following is the list of dependencies you need.

- Struts version 1.1 or higher
- Commons BeanUtils
- Commons Digester
- Commons Lang
- Commons Logging

These dependencies are all available in the Spring distribution.

14.3.2. How to integrate Tiles

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at http://jakarta.apache.org/struts). In Spring this is done using the TilesConfigurer. Have a look at the following piece of example ApplicationContext configuration:

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="factoryClass" value="org.apache.struts.tiles.xmlDefinition.I18nFactorySet"/>
```

```
 <property name="definitions">
  <list>
   <value>/WEB-INF/defs/general.xml</value>
   <value>/WEB-INF/defs/widgets.xml</value>
   <value>/WEB-INF/defs/administrator.xml</value>
   <value>/WEB-INF/defs/customer.xml</value>
   <value>/WEB-INF/defs/templates.xml</value>
  </list>
 </property>
</bean>
```

As you can see, there are five files containing definitions, which are all located in the WEB-INF/defs directory. At initialization of the WebApplicationContext, the files will be loaded and the definitionsfactory defined by thefactoryClass-property is initialized. After that has been done, the tiles includes in the definition files can be used as views within your Spring web application. To be able to use the views you have to have a ViewResolver just as with any other view technology used with Spring. Below you can find two possibilities, the InternalResourceViewResolver and the ResourceBundleViewResolver.

14.3.2.1. InternalResourceViewResolver

The InternalResourceViewResolver instantiates the given viewClass for each view it has to resolve.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="requestContextAttribute" value="requestContext"/>
 <property name="viewClass" value="org.springframework.web.servlet.view.tiles.TilesView"/>
</bean>
```

14.3.2.2. ResourceBundleViewResolver

The ResourceBundleViewResolver has to be provided with a property file containing viewnames and viewclasses the resolver can use:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
 <property name="basename" value="views"/>
</bean>
   ...
welcomeView.class=org.springframework.web.servlet.view.tiles.TilesView
welcomeView.url=welcome (<b>this is the name of a definition</b>)
vetsView.class=org.springframework.web.servlet.view.tiles.TilesView
vetsView.url=vetsView (<b>again, this is the name of a definition</b>)
findOwnersForm.class=org.springframework.web.servlet.view.JstlView
findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
...
```

As you can see, when using the ResourceBundleViewResolver, you can mix view using different view technologies.

14.4. Velocity & FreeMarker

Velocity and FreeMarker are two templating languages that can both be used as view technologies within Spring MVC applications. The languages are quite similar and serve similar needs and so are considered together in this section. For semantic and syntactic differences between the two languages, see the FreeMarker web site.

14.4.1. Dependencies

Your web application will need to include velocity-1.x.x.jar or freemarker-2.x.jar in order to work with Velocity or FreeMarker respectively and commons-collections.jar needs also to be available for Velocity. Typically they are included in the WEB-INF/lib folder where they are guaranteed to be found by a J2EE server and added to the classpath for your application. It is of course assumed that you already have the spring.jar in your WEB-INF/lib folder too! The latest stable velocity, freemarker and commons collections jars are supplied with the Spring framework and can be copied from the relevant /lib/ sub-directories. If you make use of Spring's dateToolAttribute or numberToolAttribute in your Velocity views, you will also need to include the velocity-tools-generic-1.x.jar

14.4.2. Context configuration

A suitable configuration is initialized by adding the relevant configurer bean definition to your *-servlet.xml as shown below:

```
<!--
  This bean sets up the Velocity environment for us based on a root path for templates.
  Optionally, a properties file can be specified for more control over the Velocity
  environment, but the defaults are pretty sane for file based template loading.
-->
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<!--
  View resolvers can also be configured with ResourceBundles or XML files. If you need
  different view resolving based on Locale, you have to use the resource bundle resolver.
-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".vm"/>
</bean>
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!--
  View resolvers can also be configured with ResourceBundles or XML files. If you need
  different view resolving based on Locale, you have to use the resource bundle resolver.
-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
<property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>
</bean>
```

NB: For non web-apps add a VelocityConfigurationFactoryBean or a FreeMarkerConfigurationFactoryBean to your application context definition file.

14.4.3. Creating templates

Your templates need to be stored in the directory specified by the *Configurer bean shown above in Section 14.4.2, “Context configuration” This document does not cover details of creating templates for the two languages - please see their relevant websites for information. If you use the view resolvers highlighted, then the logical view names relate to the template file names in similar fashion to InternalResourceViewResolver for JSP's. So if your controller returns a ModelAndView object containing a view name of "welcome" then the resolvers will look for the /WEB-INF/freemarker/welcome.ftl or /WEB-INF/velocity/welcome.vm template as appropriate.

14.4.4. Advanced configuration

The basic configurations highlighted above will be suitable for most application requirements, however additional configuration options are available for when unusual or advanced requirements dictate.

14.4.4.1. velocity.properties

This file is completely optional, but if specified, contains the values that are passed to the Velocity runtime in order to configure velocity itself. Only required for advanced configurations, if you need this file, specify its location on the VelocityConfigurer bean definition above.

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="configLocation value="/WEB-INF/velocity.properties"/>
</bean>
```

Alternatively, you can specify velocity properties directly in the bean definition for the Velocity config bean by replacing the "configLocation" property with the following inline properties.

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
 <property name="velocityProperties">
  <props>
   <prop key="resource.loader">file</prop>
   <prop key="file.resource.loader.class">
    org.apache.velocity.runtime.resource.loader.FileResourceLoader
   </prop>
   <prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>
   <prop key="file.resource.loader.cache">false</prop>
  </props>
 </property>
</bean>
```

Refer to the API documentation for Spring configuration of Velocity, or the Velocity documentation for examples and definitions of the velocity.properties file itself.

14.4.4.2. FreeMarker

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker Configuration object managed by Spring by setting the appropriate bean properties on the FreeMarkerConfigurer bean. The freemarkerSettings property requires a java.util.Properties object and the freemarkerVariables property requires a java.util.Map.

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
 <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
 <property name="freemarkerVariables">
  <map>
   <entry key="xml_escape" ref="fmXmlEscape"/>
  </map>
 </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

See the FreeMarker documentation for details of settings and variables as they apply to the Configuration object.

14.4.5. Bind support and form handling

Spring provides a tag library for use in JSP's that contains (amongst other things) a <spring:bind> tag. This tag primarily enables forms to display values from form backing objects and to show the results of failed validations from aValidator in the web or business tier. From version 1.1, Spring now has support for the same functionality in both Velocity and FreeMarker, with additional convenience macros for generating form input elements themselves.

14.4.5.1. the bind macros

A standard set of macros are maintained within the spring.jar file for both languages, so they are always available to a suitably configured application. However they can only be used if your view sets the bean propertyexposeSpringMacroHelpers to true . The same property can be set on VelocityViewResolver or FreeMarkerViewResolvertoo if you happen to be using it, in which case all of your views will inherit the value from it. Note that this property is **not required** for any aspect of HTML form handling **except** where you wish to take advantage of the Spring macros. Below is an example of a view.properties file showing correct configuration of such a view for either language;

```
personFormV.class=org.springframework.web.servlet.view.velocity.VelocityView
personFormV.url=personForm.vm
personFormV.exposeSpringMacroHelpers=true
personFormF.class=org.springframework.web.servlet.view.freemarker.FreeMarkerView
personFormF.url=personForm.ftl
personFormF.exposeSpringMacroHelpers=true
```

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections concentrate

only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the files are called spring.vm / spring.ftl and are in the packages org.springframework.web.servlet.view.velocity ororg.springframework.web.servlet.view.freemarker respe ctively.

14.4.5.2. simple binding

In your html forms (vm / ftl templates) that act as the 'formView' for a Spring form controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Note that the name of the command object is "command" by default, but can be overridden in your MVC configuration by setting the 'commandName' bean property on your form controller. Example code is shown below for thepersonFormV and personFormF views configured earlier;

```
<!-- velocity macros are automatically available -->
<html>
...
<form action="" method="POST">
  Name:
  #springBind( "command.name" )
  <input type="text"
    name="${status.expression}"
    value="$!status.value" /><br>
  #foreach($error in $status.errorMessages) <b>$error</b> <br> #end
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
<!-- freemarker macros have to be imported into a namespace.  We strongly
recommend sticking to 'spring' -->
<#import "spring.ftl" as spring />
<html>
...
<form action="" method="POST">
  Name:
  <@spring.bind "command.name" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>
  ...
  <input type="submit" value="submit"/>
</form>...
</html>
```

#springBind / <@spring.bind> requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The bind macro assumes the default HTML escaping behavior specified by the ServletContext parameter defaultHtmlEscape in web.xml

The optional form of the macro called #springBindEscaped / <@spring.bindEscaped> takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

14.4.5.3. form input generation macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

The following table of available macros show the VTL and FTL definitions and the parameter list that each takes.

Table 14.1. table of macro definitions

macro	VTL definition	FTL definition
message (output a string from a resource bundle based on the code parameter)	#springMessage($code)	<@spring.message code/>
messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)	#springMessageText($code $default)	<@spring.messageText code, default/>
url (prefix a relative URL with the application's context root)	#springUrl($relativeUrl)	<@spring.url relativeUrl/>
formInput (standard input field for gathering user input)	#springFormInput($path $attributes)	<@spring.formInput path, attributes, fieldType/>
formHiddenInput * (hidden input field for submitting non-user input)	#springFormHiddenInput($path $attributes)	<@spring.formHiddenInput path, attributes/>
formPasswordInput * (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type)	#springFormPasswordInput($path $attributes)	<@spring.formPasswordInput path, attributes/>
formTextarea (large text field for gathering long, freeform text input)	#springFormTextarea($path $attributes)	<@spring.formTextarea path, attributes/>
formSingleSelect (drop down box of options allowing a single required value to be selected)	#springFormSingleSelect($path $options $attributes)	<@spring.formSingleSelect path, options, attributes/>
formMultiSelect (a list box of options allowing the user to select 0 or more values)	#springFormMultiSelect($path $options $attributes)	<@spring.formMultiSelect path, options, attributes/>
formRadioButtons (a set of radio buttons allowing a single selection to be made from the available choices)	#springFormRadioButtons($path $options $separator $attributes)	<@spring.formRadioButtons path, options separator, attributes/>
formCheckboxes (a set of checkboxes allowing 0 or more values to be selected)	#springFormCheckboxes($path $options $separator $attributes)	<@spring.formCheckboxes path, options, separator, attributes/>
showErrors (simplify display of validation errors for the bound field)	#springShowErrors($separator $classOrStyle)	<@spring.showErrors separator, classOrStyle/>

* In FTL (FreeMarker), these two macros are not actually required as you can use the normal formInput macro, specifying 'hidden' or 'password' as the value for the fieldType parameter.

The parameters to any of the above macros have consistent meanings:

- path: the name of the field to bind to (ie "command.name")
- options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from

the corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a SortedMap such as aTreeMap with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a LinkedHashMap or a LinkedMap from commons-collections.

- separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "
").
- attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.
- classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in tags.

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

14.4.5.3.1. Input Fields

```
<!-- the Name field example from above using form macros in VTL -->
...
  Name:
  #springFormInput("command.name" "")<br>
  #springShowErrors("<br>" "")<br>
```

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn't need to pass the path parameter again - it simply operates on whichever field a bind was last created for.
The showErrors macro takes a separator parameter (the characters that will be used to separate multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter, unlike Velocity, and the two macro calls above could be expressed as follows in FTL:

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

Output is shown below of the form fragment generating the name field, and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring's Validation framework.
The generated HTML looks like this:

```
Name:
 <input type="text" name="name" value=""
>
<br>
 <b>required</b>
<br>
<br>
```

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

14.4.5.3.2. Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.
An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

```
...
 Town:
 <@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

This renders a line of radio buttons, one for each value in cityMap using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the same String for each key-value pair in the map. The map's keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

```
Town:
<input type="radio" name="address.town" value="London"

>
London
<input type="radio" name="address.town" value="Paris"
 checked="checked"
>
Paris
<input type="radio" name="address.town" value="New York"

>
New York
```

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

```
protected Map referenceData(HttpServletRequest request) throws Exception {
 Map cityMap = new LinkedHashMap();
 cityMap.put("LDN", "London");
 cityMap.put("PRS", "Paris");
 cityMap.put("NYC", "New York");
 Map m = new HashMap();
 m.put("cityMap", cityMap);
 return m;
}
```

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

```
Town:
<input type="radio" name="address.town" value="LDN"

>
London
<input type="radio" name="address.town" value="PRS"
 checked="checked"
>
Paris
<input type="radio" name="address.town" value="NYC">
New York
```

14.4.5.4. Overriding HTML escaping and making tags XHTML compliant

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring's bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of 'true' for a model/context variable named xhtmlCompliant:

```
## for Velocity..
#set($springXhtmlCompliant = true)
<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.
In similar fashion, HTML escaping can be specified per field:

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name" />

<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

14.5. XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring application.

14.5.1. My First Words

This example is a trivial Spring application that creates a list of words in the Controller and adds them to the model map. The map is returned along with the view name of our XSLT view. See Section 13.3, “Controllers” for details of Spring Controllers. The XSLT view will turn the list of words into a simple XML document ready for transformation.

14.5.1.1. Bean definitions

Configuration is standard for a simple Spring application. The dispatcher servlet config file contains a reference to aViewResolver, URL mappings and a single controller bean..

```
<bean id="homeController"class="xslt.HomeController"/>
```

..that implements our word generation 'logic'.

14.5.1.2. Standard MVC controller code

The controller logic is encapsulated in a subclass of AbstractController, with the handler method being defined like so..

```
protected ModelAndView handleRequestInternal(
    HttpServletRequest req, HttpServletResponse resp) throws Exception {
    Map map = new HashMap();
    List wordList = new ArrayList();

    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

So far we've done nothing that's XSLT specific. The model data has been created in the same way as you would for any other Spring MVC application. Depending on the configuration of the application now, that list of words could be rendered by JSP/JSTL by having them added as request attributes, or they could be handled by Velocity by adding the object to the VelocityContext. In order to have XSLT render them, they of course have to be converted into an XML document somehow. There are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This

prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

14.5.1.3. Convert the model data to XML

In order to create a DOM document from our list of words or any other model data, we subclassorg.springframework.web.servlet.view.xslt.AbstractXsltView. In doing so, we must implement the abstract methodcreateDomNode(). The first parameter passed to this method is our model Map. Here's the complete listing of the HomePage class in our trivial word application - it uses JDOM to build the XML document before converting it to the required W3C Node, but this is simply because I find JDOM (and Dom4J) easier API's to handle than the W3C API.

```
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

   protected Node createDomNode(
      Map model, String rootName, HttpServletRequest req, HttpServletResponse res
   ) throws Exception {

      org.jdom.Document doc = new org.jdom.Document();
      Element root = new Element(rootName);
      doc.setRootElement(root);

      List words = (List) model.get("wordList");
      for (Iterator it = words.iterator(); it.hasNext();) {
         String nextWord = (String) it.next();
         Element e = new Element("word");
         e.setText(nextWord);
         root.addContent(e);
      }

      // convert JDOM doc to a W3C Node and return
      return new DOMOutputter().output( doc );
   }

}
```

14.5.1.3.1. Adding stylesheet parameters

A series of parameter name/value pairs can optionally be defined by your subclass which will be added to the transformation object. The parameter names must match those defined in your XSLT template declared with <xsl:param name="myParam">defaultValue</xsl:param> To specify the parameters, override the method getParameters() from AbstractXsltView and return a Map of the name/value pairs. If your parameters need to derive information from the current request, you can (from version 1.1) override the getParameters(HttpServletRequest request) method instead.

14.5.1.3.2. Formatting dates and currency

Unlike JSTL and Velocity, XSLT has relatively poor support for locale based currency and date formatting. In recognition of the fact, Spring provides a helper class that you can use from within your createDomNode() methods to get such support. See the javadocs for org.springframework.web.servlet.view.xslt.FormatHelper

14.5.1.4. Defining the view properties

The views.properties file (or equivalent xml definition if you're using an XML based view resolver as we did in the Velocity examples above) looks like this for the one-view application that is 'My First Words'..

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words
```

Here, you can see how the view is tied in with the HomePage class just written which handles the model domification in the first property '.class'. The stylesheetLocation property obviously points to the XSLT file which will handle the XML transformation into HTML for us and the final property '.root' is the name that will be used as the root of the XML document. This gets passed to the HomePage class above in the second parameter to the createDomNode method.

14.5.1.5. Document transformation

Finally, we have the XSLT code used for transforming the above document. As highlighted in the views.properties file, it is called home.xslt and it lives in the war file under WEB-INF/xsl.

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text/html" omit-xml-declaration="yes"/>
  <xsl:template match="/">
    <html>
      <head><title>Hello!</title></head>
      <body>
        <h1>My First Words</h1>
        <xsl:for-each select="wordList/word">
          <xsl:value-of select="."/><br />
        </xsl:for-each>

      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

14.5.2. Summary

A summary of the files discussed and their location in the WAR file is shown in the simplified WAR structure below.

```
ProjectRoot
 |
 +- WebContent
    |
    +- WEB-INF
       |
       +- classes
       |   |
       |   +- xslt
       |   |   |
       |   |   +- HomePageController.class
       |   |   +- HomePage.class
       |   |
       |   +- views.properties
       |
       +- lib
       |   |
       |   +- spring.jar
       |
       +- xsl
       |   |
       |   +- home.xslt
       |
       +- frontcontroller-servlet.xml
```

You will also need to ensure that an XML parser and an XSLT engine are available on the classpath. JDK 1.4 provides them by default, and most J2EE containers will also make them available by default, but it's a possible source of errors to be aware of.

14.6. Document views (PDF/Excel)

14.6.1. Introduction

Returning an HTML page isn't always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.
In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the iText.jar. Both are included in the main Spring distribution.

14.6.2. Configuration and setup

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

14.6.2.1. Document view definitions

Firstly, let's amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier..

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.class=excel.HomePage

pdf.class=pdf.HomePage
```

If you want to start with a template spreadsheet to add your model data to, specify the location as the 'url' property in the view definition

14.6.2.2. Controller code

The controller code we'll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

14.6.2.3. Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass oforg.springframework.web.servlet.view.document.AbstractExcelView (for Excel files generated by POI) or org.springframework.web.servlet.view.document.AbstractJExcelView (for JExcelApi-generated Excel files). and implementing the buildExcelDocument
Here's the complete listing for our POI Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet..

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

   protected void buildExcelDocument(
      Map model,
      HSSFWorkbook wb,
      HttpServletRequest req,
      HttpServletResponse resp)
      throws Exception {
```

```
        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        //sheet = wb.getSheetAt( 0 );
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short)12);

        // write a text at A1
        cell = getCell( sheet, 0, 0 );
        setText(cell,"Spring-Excel test");

        List words = (List ) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell( sheet, 2+i, 0 );
            setText(cell, (String) words.get(i));

        }
    }
}
```

And this a view generating the same Excel file, now using JExcelApi:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model,
        WritableWorkbook wb,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {

        WritableSheet sheet = wb.createSheet("Spring");

        sheet.addCell(new Label(0, 0, "Spring-Excel test");
        List words  = (List)model.get("wordList");
        for (int i = -; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String)words.get(i));
        }
    }
}
```

Note the differences between the APIs. We've found that the JExcelApi is somewhat more intuitive and furthermore, JExcelApi has a bit better image-handling capabilities. There have been memory problems with large Excel file when using JExcelApi however.

If you now amend the controller such that it returns xl as the name of the view (return new ModelAndView("xl", map);) and run your application again, you should find that the Excel spreadsheet is created and downloaded automatically when you request the same page as before.

14.6.2.4. Subclassing for PDF views

The PDF version of the word list is even simpler. This time, the class extendsorg.springframework.web.servlet.view.document.AbstractPdfView and implements the buildPdfDocument() method as follows..

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

   protected void buildPdfDocument(Map model,  Document doc,  PdfWriter writer, HttpServletRequest req,
      HttpServletResponse resp)  throws Exception {

      List words = (List) model.get("wordList");
        for (int i=0; i<words.size(); i++)
          doc.add( new Paragraph((String) words.get(i)));

   }
}
```

Once again, amend the controller to return the pdf view with a return new ModelAndView("pdf", map); and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

14.7. JasperReports

JasperReports (http://jasperreports.sourceforge.net) is a powerful, open-source reporting engine that supports the creation of report designs using an easily understood XML file formats. JasperReports is capable of rendering reports output into four different formats: CSV, Excel, HTML and PDF.

14.7.1. Dependencies

Your application will need to include the latest release of JasperReports, which at the time of writing was 0.6.1. JasperReports itself depends on the following projects:

- BeanShell
- Commons BeanUtils
- Commons Collections
- Commons Digester
- Commons Logging
- iText
- POI

JasperReports also requires a JAXP compliant XML parser.

14.7.2. Configuration

To configure JasperReports views in your ApplicationContext you have to define a ViewResolver to map view names to the appropriate view class depending on which format you want your report rendered in.

14.7.2.1. Configuring the ViewResolver

Typically, you will use the ResourceBundleViewResolver to map view names to view classes and files in a properties file

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
   <property name="basename" value="views"/>
</bean>
```

Here we've configured an instance of ResourceBundleViewResolver which will look for view mappings in the resource bundle with base name views. The exact contents of this file is described in the next section.

14.7.2.2. Configuring the Views

Spring contains five different View implementations for JasperReports four of which corresponds to one of the four output formats supported by JasperReports and one that allows for the format to be determined at runtime:

Table 14.2. JasperReports View Classes

Class Name	Render Format
JasperReportsCsvView	CSV
JasperReportsHtmlView	HTML

Class Name	Render Format
JasperReportsPdfView	PDF
JasperReportsXlsView	Microsoft Excel
JasperReportsMultiFormatView	Decided at runtime (see Section 14.7.2.4, "Using JasperReportsMultiFormatView")

Mapping one of these classes to a view name and a report file is simply a matter of adding the appropriate entries into the resource bundle configured in the previous section as shown here:

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
```

Here you can see that the view with name, simpleReport, is mapped to the JasperReportsPdfView class. This will cause the output of this report to be rendered in PDF format. The url property of the view is set to the location of the underlying report file.

14.7.2.3. About Report Files

JasperReports has two distinct types of report file: the design file, which has a .jrxml extension, and the compiled report file, which has a .jasper extension. Typically, you use the JasperReports Ant task to compile your .jrxml design file into a .jasper file before deploying it into your application. With Spring you can map either of these files to your report file and Spring will take care of compiling the .jrxml file on the fly for you. You should note that after a .jrxml file is compiled by Spring, the compiled report is cached for the life of the application. To make changes to the file you will need to restart your application.

14.7.2.4. Using JasperReportsMultiFormatView

The JasperReportsMultiFormatView allows for report format to be specified at runtime. The actual rendering of the report is delegated to one of the other JasperReports view classes - the JasperReportsMultiFormatView class simply adds a wrapper layer that allows for the exact implementation to be specified at runtime.
The JasperReportsMultiFormatView class introduces two concepts: the format key and the discriminator key. TheJasperReportsMultiFormatView class uses the mapping key to lookup the actual view implementation class and uses the format key to lookup up the mapping key. From a coding perspective you add an entry to your model with the formay key as the key and the mapping key as the value, for example:

```
public ModelAndView handleSimpleReportMulti(HttpServletRequest request,
HttpServletResponse response) throws Exception {
  String uri = request.getRequestURI();
  String format = uri.substring(uri.lastIndexOf(".") + 1);
  Map model = getModel();
  model.put("format", format);
  return new ModelAndView("simpleReportMulti", model);
}
```

In this example, the mapping key is determined from the extension of the request URI and is added to the model under the default format key: format. If you wish to use a different format key then you can configure this using the formatKeyproperty of the JasperReportsMultiFormatView class.
By default the following mapping key mappings are configured in JasperReportsMultiFormatView:

Table 14.3. JasperReportsMultiFormatView Default Mapping Key Mappings

Mapping Key	View Class
csv	JasperReportsCsvView
html	JasperReportsHtmlView
pdf	JasperReportsPdfView
xls	JasperReportsXlsView

So in the example above a request to URI /foo/myReport.pdf would be mapped to the JasperReportsPdfView class. You can override the mapping key to view class mappings using the formatMappings property of JasperReportsMultiFormatView.

14.7.3. Populating the ModelAndView

In order to render your report correctly in the format you have chosen, you must supply Spring with all of the data needed to populate your report. For JasperReports this means you must pass in all report parameters along with the report datasource. Report parameters are simple name/value pairs and can be added be to the Map for your model as you would add any name/value pair.

When adding the datasource to the model you have two approaches to choose from. The first approach is to add an instance of JRDataSource or Collection to the model Map under any arbitrary key. Spring will then locate this object in the model and treat it as the report datasource. For example, you may populate your model like this:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  model.put("myBeanData", beanData);
  return model;
}
```

The second approach is to add the instance of JRDataSource or Collection under a specific key and then configure this key using the reportDataKey property of the view class. In both cases Spring will instances of Collection in aJRBeanCollectionDataSource instance. For example:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  Collection someData = getSomeData();
  model.put("myBeanData", beanData);
  model.put("someData", someData);
  return model;
}
```

Here you can see that two Collection instances are being added to the model. To ensure that the correct one is used, we simply modify our view configuration as appropriate:

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData
```

Be aware that when using the first approach, Spring will use the first instance of JRDataSource or Collection that it encounters. If you need to place multiple instances of JRDataSource or Collection into the model then you need to use the second approach.

14.7.4. Working with Sub-Reports

JasperReports provides support for embedded sub-reports within your master report files. There are a wide variety of mechanisms for including sub-reports in your report files. The easiest way is to hard code the report path and the SQL query for the sub report into your design files. The drawback of this approach is obvious - the values are hard-coded into your report files reducing reusability and making it harder to modify and update report designs. To overcome this you can configure sub-reports declaratively and you can include additional data for these sub-reports directly from your controllers.

14.7.4.1. Configuring Sub-Report Files

To control which sub-report files are included in a master report using Spring, your report file must be configured to accept sub-reports from an external source. To do this you declare a parameter in your report file like this:

```
<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>
```

Then, you define your sub-report to use this sub-report parameter:

```
<subreport>
   <reportElement isPrintRepeatedValues="false" x="5" y="25" width="325"
```

```
    height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>
  <subreportParameter name="City">
    <subreportParameterExpression><![CDATA[$F{city}]]></subreportParameterExpression>
  </subreportParameter>
  <dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpression>
  <subreportExpression class="net.sf.jasperreports.engine.JasperReport">
        <![CDATA[$P{ProductsSubReport}]]></subreportExpression>
</subreport>
```

This defines a master report file that expects the sub-report to be passed in as an instance ofnet.sf.jasperreports.engine.JasperReports under the parameter ProductsSubReport. When configuring your Jasper view class, you can instruct Spring to load a report file and pass into the JasperReports engine as a sub-report using thesubReportUrls property:

```
<property name="subReportUrls">
  <map>
    <entry key="ProductsSubReport" value="/WEB-INF/reports/subReportChild.jrxml"/>
  </map>
</property>
```

Here, the key of the Map corresponds to the name of the sub-report parameter in th report design file, and the entry is the URL of the report file. Spring will load this report file, compiling it if necessary, and will pass into the JasperReports engine under the given key.

14.7.4.2. Configuring Sub-Report Data Sources

This step is entirely optional when using Spring configure your sub-reports. If you wish, you can still configure the data source for your sub-reports using static queries. However, if you want Spring to convert data returned in your ModelAndView into instances of JRDataSource then you need to specify which of the parameters in your ModelAndViewSpring should convert. To do this configure the list of parameter names using the subReportDataKeys property of the your chosen view class:

```
<property name="subReportDataKeys"
  value="SubReportData"/>
```

Here, the key you supply MUST correspond to both the key used in your ModelAndView and the key used in your report design file.

14.7.5. Configuring Exporter Parameters

If you have special requirements for exporter configuration - perhaps you want a specific page size for your PDF report, then you can configure these exporter parameters declaratively in your Spring configuration file using the exporterParameters property of the view class. The exporterParameters property is typed as Map and in your configuration the key of an entry should be the fully-qualified name of a static field that contains the exporter parameter definition and the value of an entry should be the value you want to assign to the parameter. An example of this is shown below:

```
<bean id="htmlReport" class="org.springframework.web.servlet.view.jasperreports.JasperReportsHtmlView">
 <property name="url" value="/WEB-INF/reports/simpleReport.jrxml"/>
 <property name="exporterParameters">
  <map>
   <entry key="net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER">
    <value>Footer by Spring!
     &lt;/td&gt;&lt;td width="50%"&gt;&amp;nbsp; &lt;/td&gt;&lt;/tr&gt;
     &lt;/table&gt;&lt;/body&gt;&lt;/html&gt;
    </value>
   </entry>
  </map>
 </property>
</bean>
```

Here you can see that the JasperReportsHtmlView is being configured with an exporter parameter fornet.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER which will output a footer in the resulting HTML.

Chapter 15. Integrating with other web frameworks

15.1. Introduction

Spring can be easily integrated into any Java-based web framework. All you need to do is to declare theContextLoaderListener in your web.xml and use a *contextConfigLocation* <context-param> to set which context files to load.

The <context-param>:

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

The <listener>:

```
<listener>
   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

NOTE: Listeners were added to the Servlet API in version 2.3. If you have a Servlet 2.2 container, you can use the ContextLoaderServlet to achieve this same functionality.

If you don't specify the *contextConfigLocation* context parameter, the ContextLoaderListener will look for a */WEB-INF/applicationContext.xml* file to load. Once the context files are loaded, Spring creates a WebApplicationContextobject based on the bean definitions and puts it into the ServletContext.

All Java web frameworks are built on top of the Servlet API, so you can use the following code to get the ApplicationContext that Spring created.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

The WebApplicationContextUtils class is for convenience, so you don't have to remember the name of the ServletContext attribute. Its *getWebApplicationContext()* method will return null if an object doesn't exist under theWebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE key. Rather than risk getting NullPointerExceptions in your application, it's better to use the *getRequiredWebApplicationContext()* method. This method throws an Exception when the *ApplicationContext* is missing.

Once you have a reference to the WebApplicationContext, you can retrieve beans by their name or type. Most developers retrieve beans by name, then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they make it easy to get beans from the BeanFactory, but they also allow you to use dependency injection on their controllers. Each framework section has more detail on its specific integration strategies.